Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Lab 4 due on 2/18
  - Homework 5 due on 2/21

Previous lecture …

- Prefix Searching Problem
- Multiple solutions:
  - DST
  - RST
  - multi-way RST

CourseMIRROR Reflections (Interesting)

- I found the idea of using an optimal number of bits for digital search trees to be interesting
- prefix symbol tables were interesting
- I found the prefix searching interesting and how it works with RSTs
- All the different tree types! Very interesting
- The applications of a radix sort trie and a digital search tree were most interesting today.
- I enjoyed learning about Radix Search Tries and how different they are.
- Different types of ADTs for searching problems such as DST, RST and large branching tries. Cool to see different ways to populate trees
- adding into DSTs and RSTs
- How a digital search tree uses only the bits not the values to place nodes

CourseMIRROR Reflections (Confusing)

- I’m not sure how the code can store the bits in the path/between the nodes.
- Also, is there no key comparison when searching a RST because you just need to check if the correct bit path exists?
- why we want to determine that it is prefix problem? We can ignore prefix and keep using RST, DST
- Runtime comparisons of the different tree types
- the thing that was confusing was radials search tree runtimes
- I would like to go over more on the run time of Digital Search Tree operations

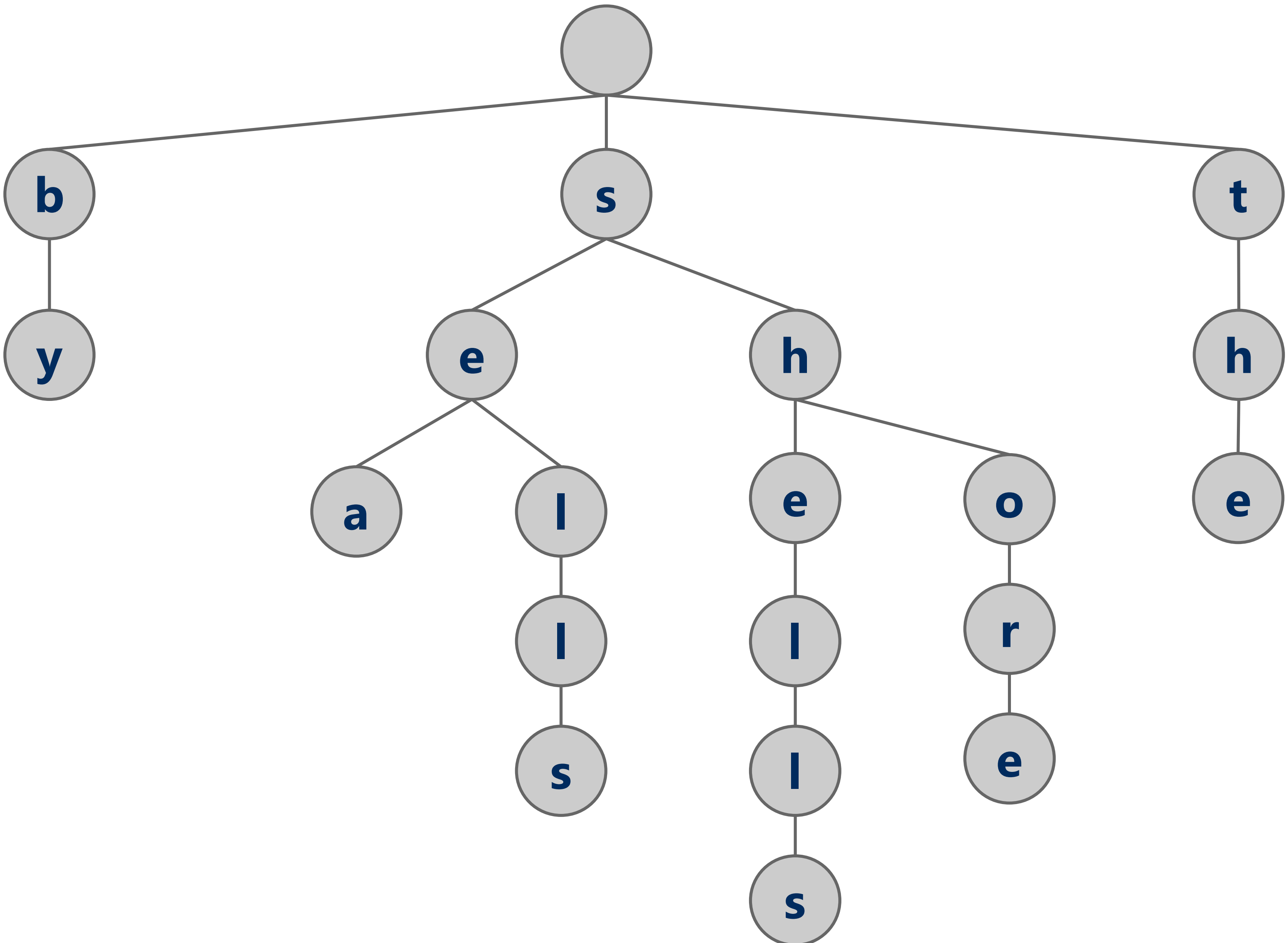
Prefix Searching (contd.)

- Input:
  - a (large) dynamic set of data items in the form of
    - *n* (key, value) pairs; key is a string from an alphabet of size *R*
    - Each key has *b* bits or *w* characters (the chars are from the alphabet)
    - What is the relationship between *b* and *w*?
  - a target *string*
- Output:
  - 0: string is not a prefix of nor equal to any of the keys
  - 1: string is a prefix of at least one key but not equal to any
  - 2: string is not a prefix of any key but is equal to a key
  - 3: string is both a prefix of at least one key and equal to one of the keys

Larger branching factor tries

- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters in a string?
  - What would this new structure look like?
- Let's try inserting the following strings into an trie:
  - she, sells, sea, shells, by, the, sea, shore

Another trie example



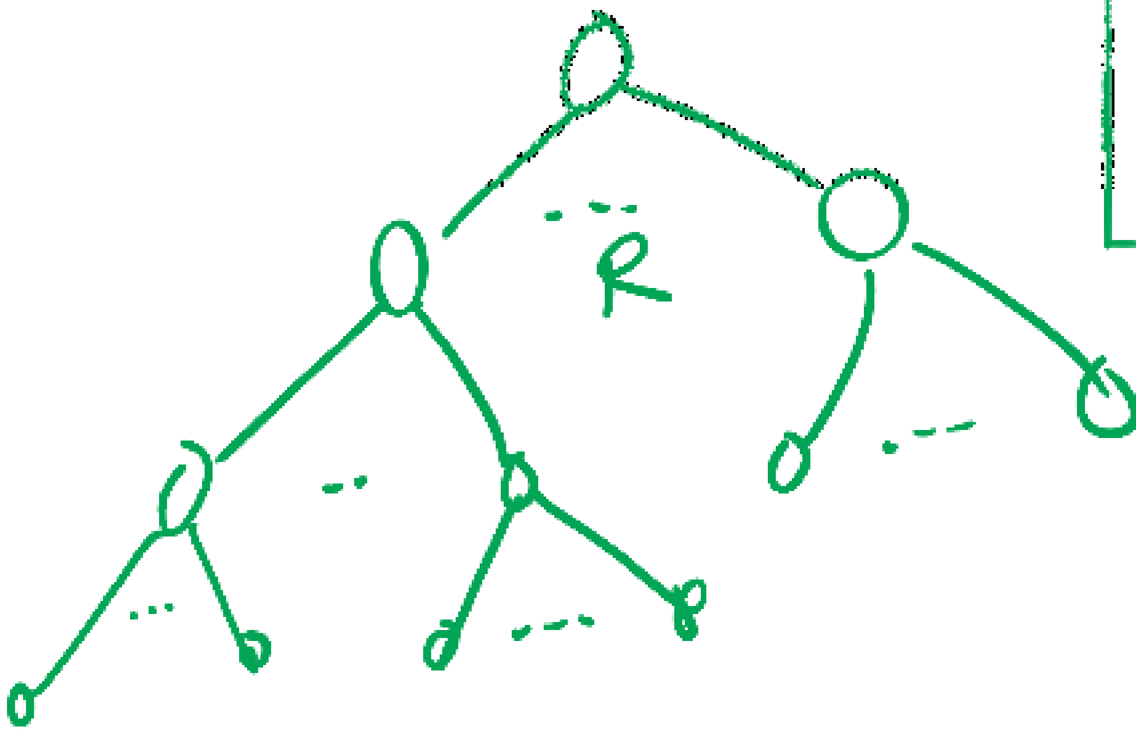
Analysis

- Runtime?

Further analysis

- Miss times
  - Require an average of $\log_R(n)$ nodes to be examined
    - Where R is the size of the alphabet being considered
    - Proof in Proposition H of Section 5.2 of the text
  - Average # of checks with 2^{20} keys in an RST?
  - With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?

Search Miss



$$\boxed{n = R^{height}}$$

$$\underline{height} = \log_R n$$

Implementation Concerns

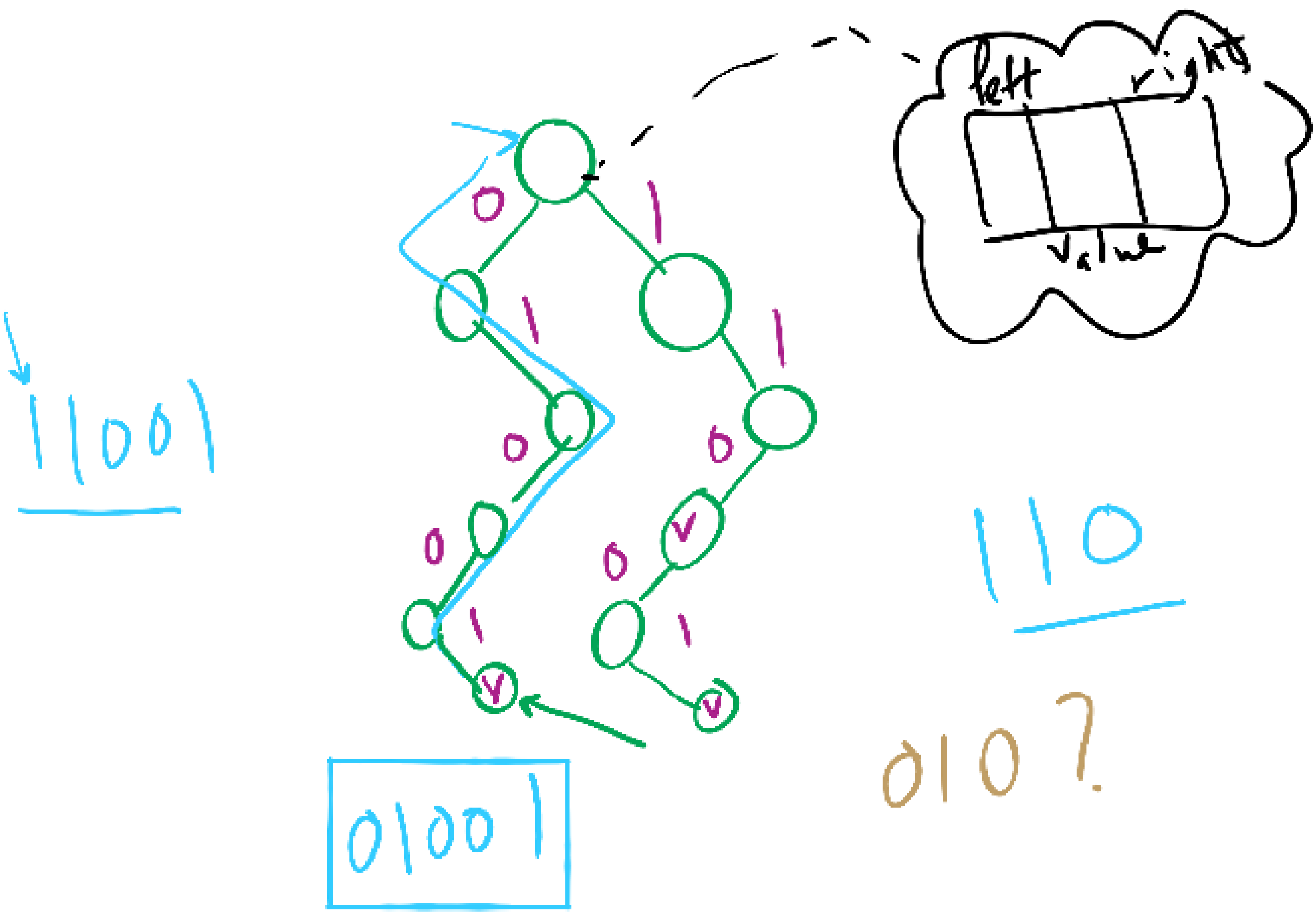
- See TrieSt.java
  - Implements an R-way trie
- Basic node object:

Where R is the branching factor

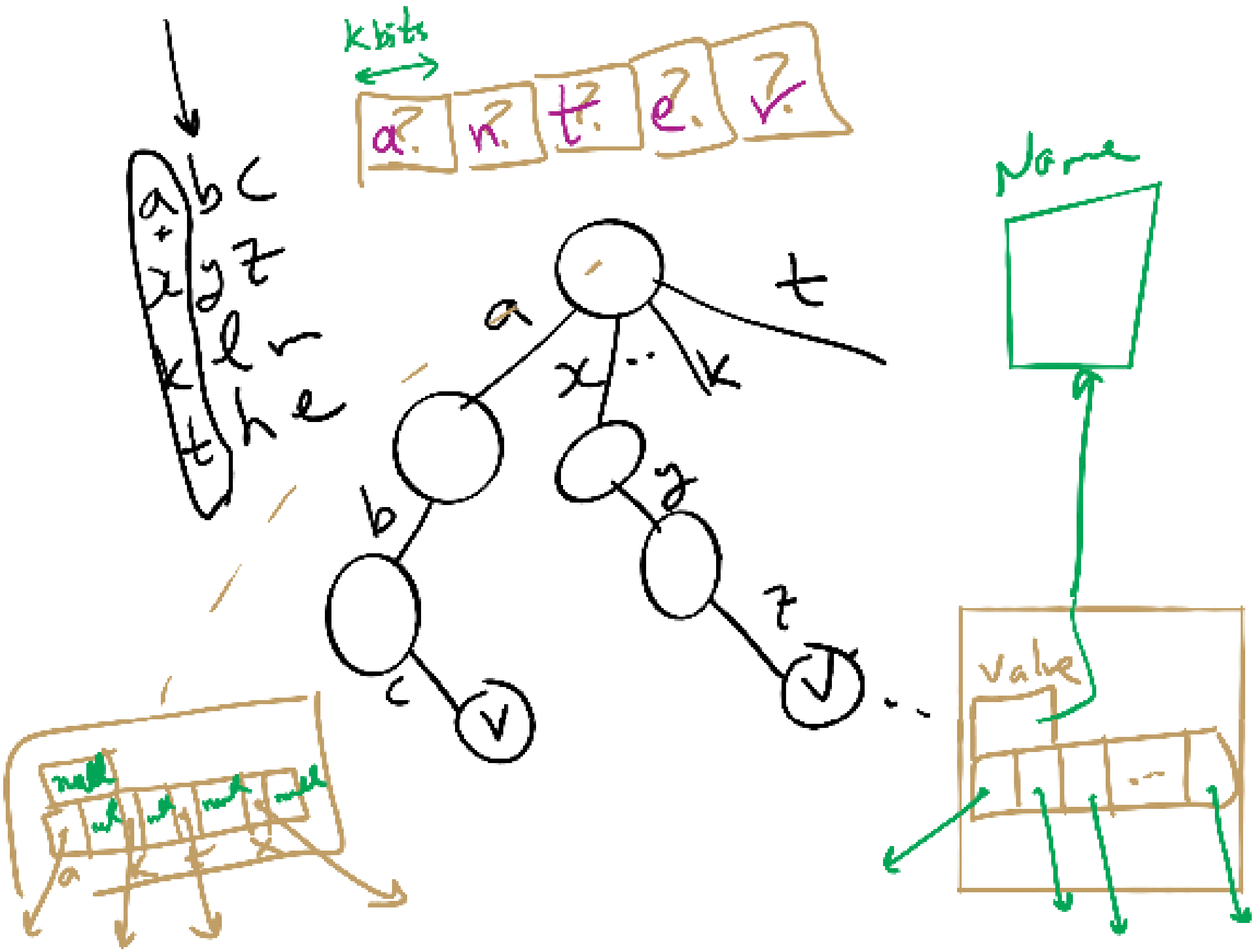
```
private static class Node {
    private Object val;
    private Node[] next = new Node[R];
}
```

- Non-null val means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

Binary RST



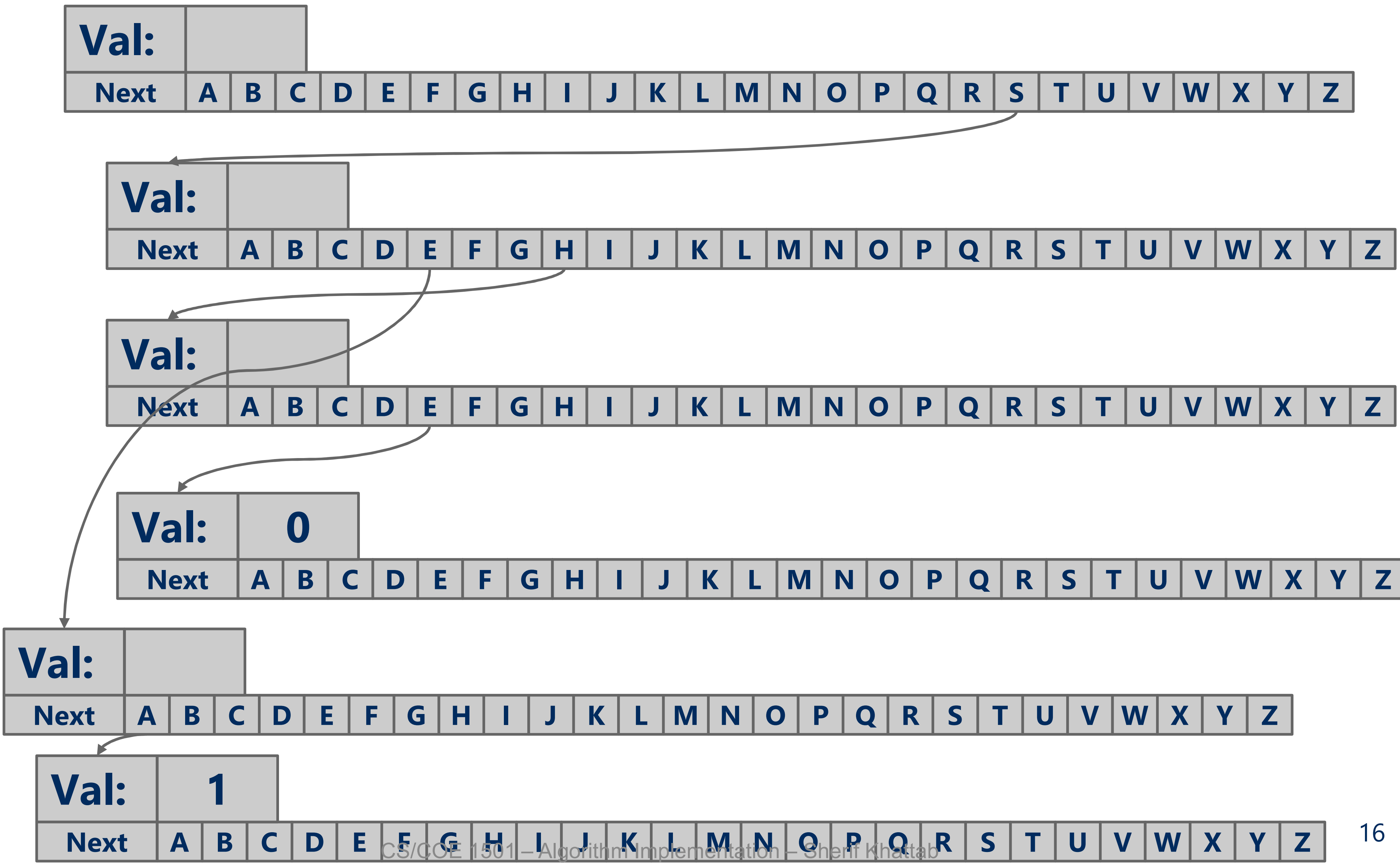
Multi-way RST



Summary of running time

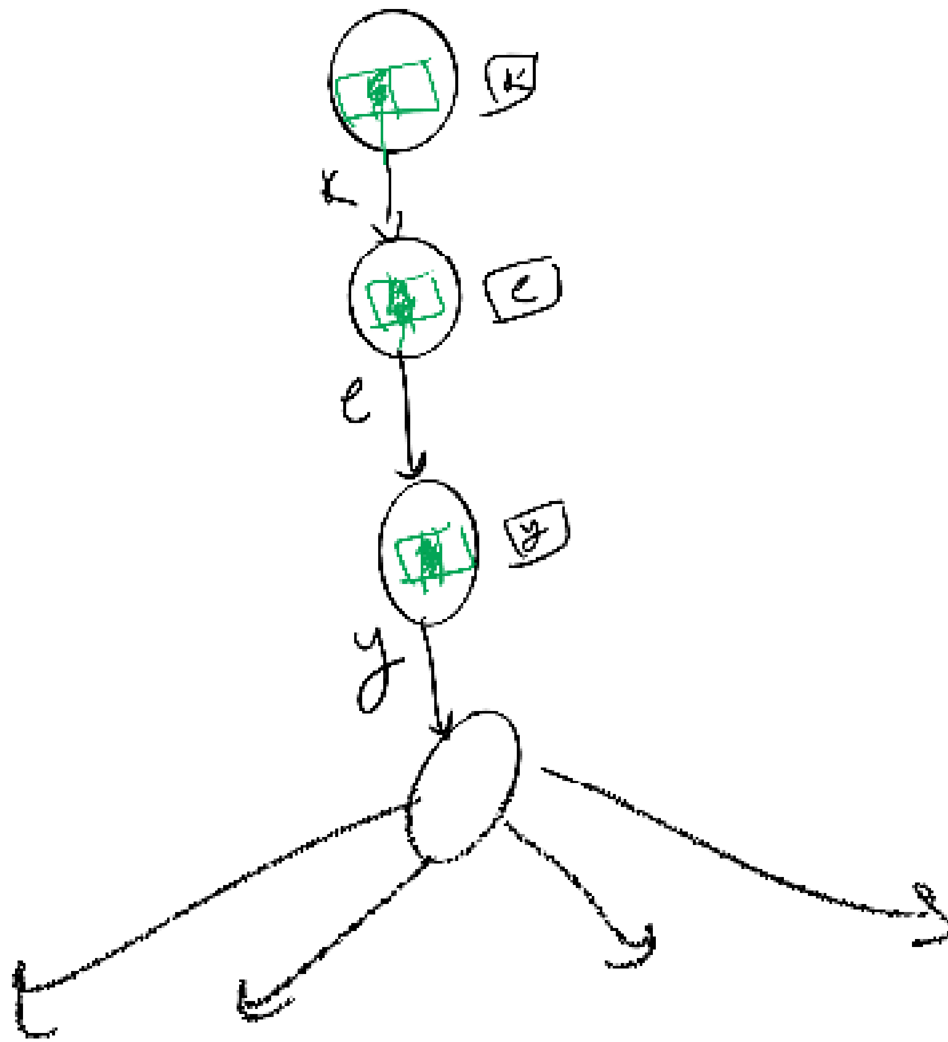
	insert	search hit	Search miss
binary BST	$\Theta(b)$	$\Theta(b)$	$\Theta(\log_2 n)$ on average
multi-way RST	$\Theta(w)$	$\Theta(w)$	$\Theta(\log_2 n)$

R-way trie example



So what’s the catch?

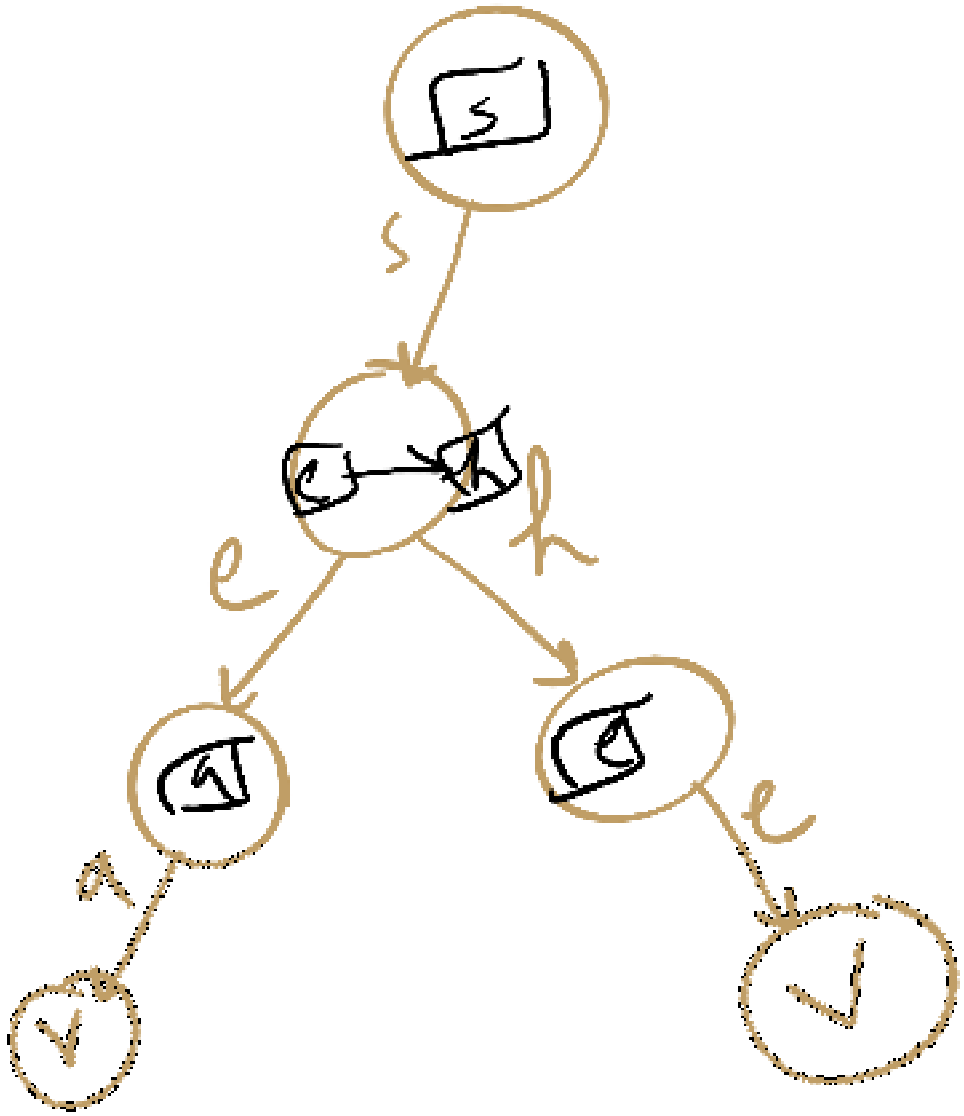
- Space!
  - Considering 8-bit ASCII, each node contains 2^8 references!
  - This is especially problematic as in many cases, alot of this space is wasted
    - Common paths or prefixes for example, e.g., if all keys begin with “key”, thats 255*3 wasted references!
    - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse



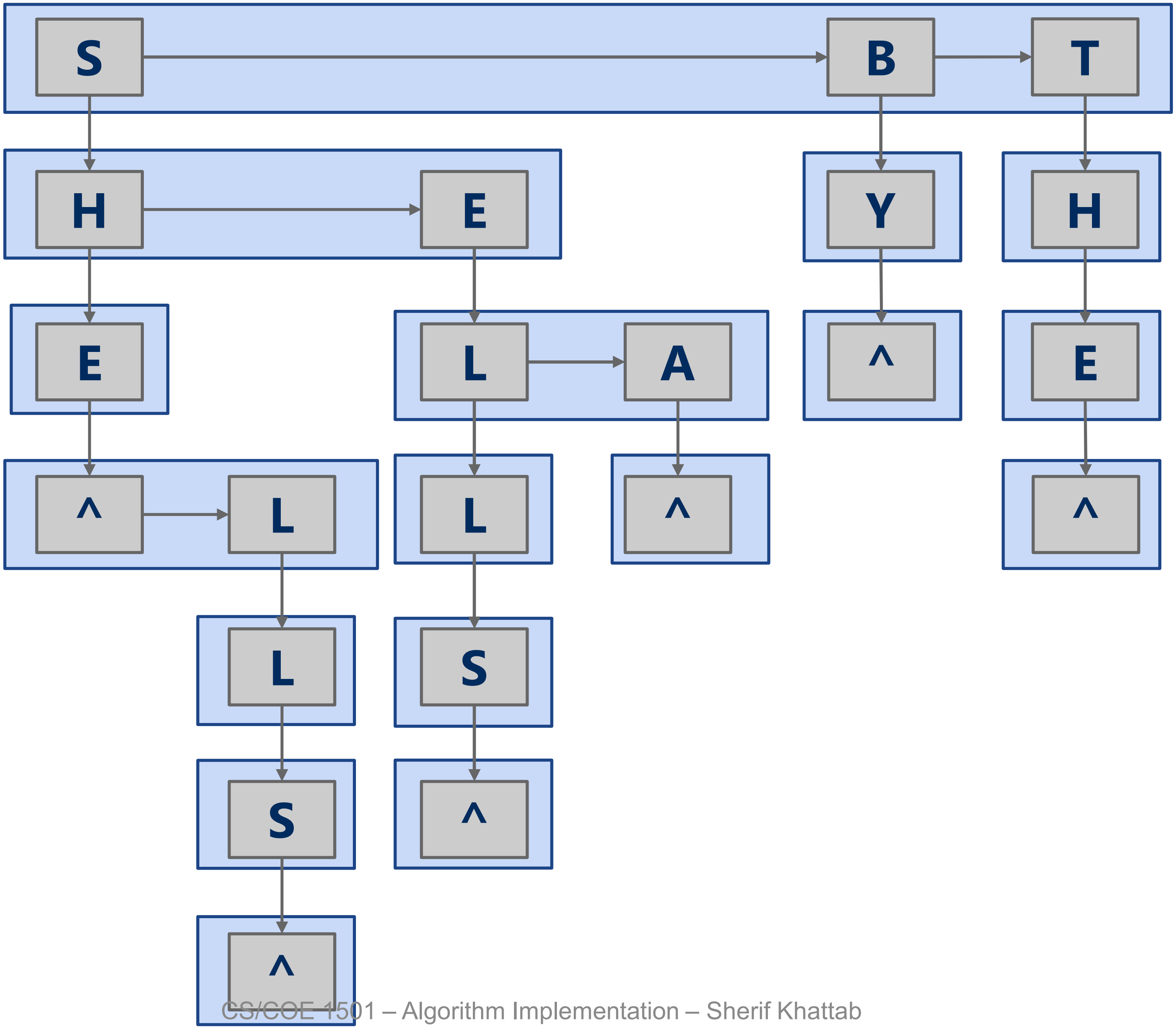
De La Briandais tries (DLBs)

- Replace the .next array of the R-way trie with a linked-list

DLB Example



DLB Example 2: nodes vs. nodelets

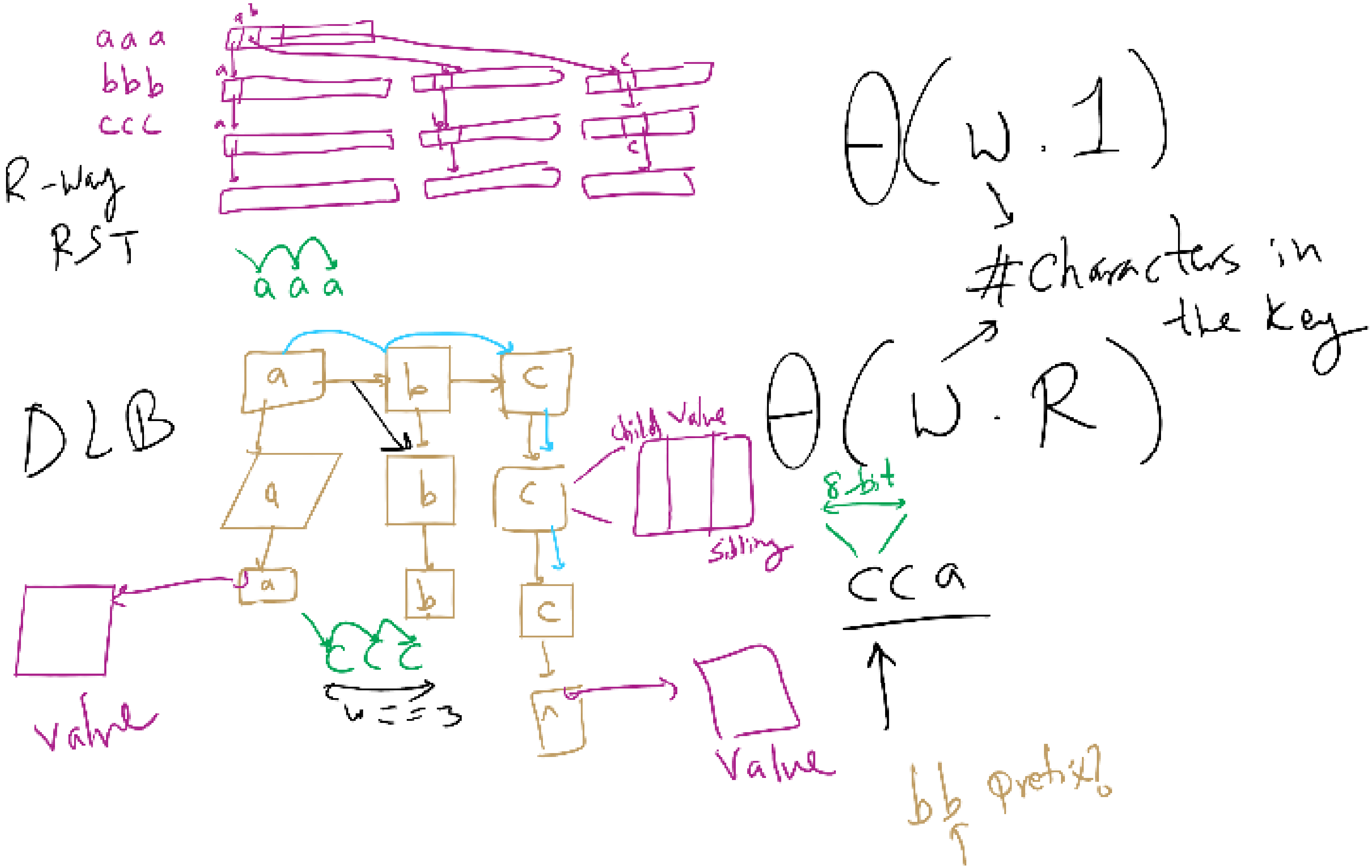


DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

	search hit insert	
R-way RST	$\Theta(w)$	
DLB	$\Theta(wR)$	

R-way RST vs. DLB



Let's go back to our Prefix Symbol Table an ADT!

- The Prefix Symbol Table ADT
  - A set of (key, value) pairs
- Operations of the PST ADT
  - insert
  - delete
  - prefixSearch
  - search
- How can we implement prefixSearch?

Runtime Comparison for Search Trees/Tries

	Search hit	Search miss (average)	insert
BST	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
RB_BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
DST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
RST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
R-way RST	$\Theta(w)$	$\Theta(\log_R n)$	$\Theta(w)$
DLB	$\Theta(wR)$	$\Theta(\log_R n \cdot R)$	$\Theta(w \cdot R)$

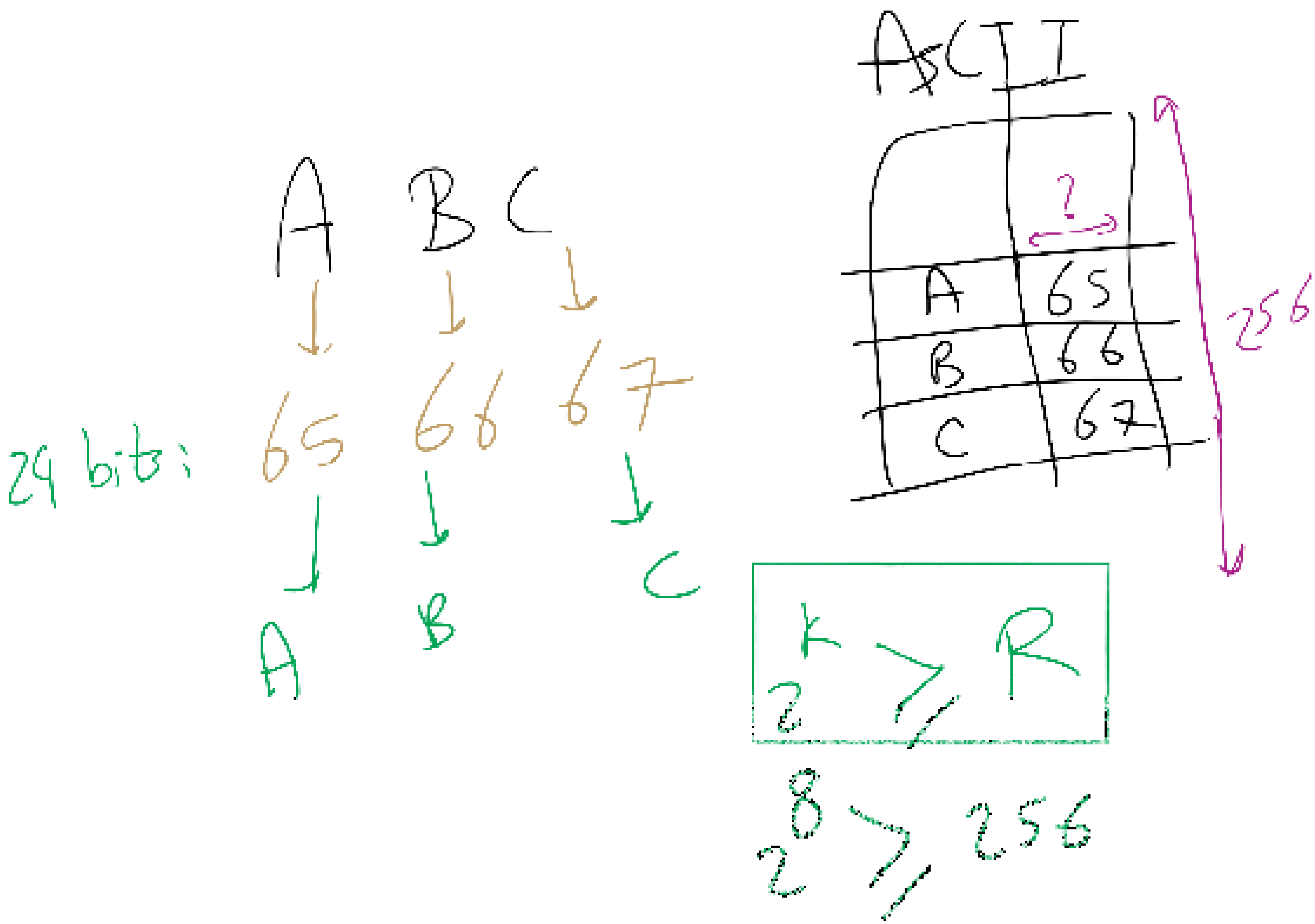
Final notes on Search Tree/Tries

- We did not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
  - Ternary search Tries
  - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text

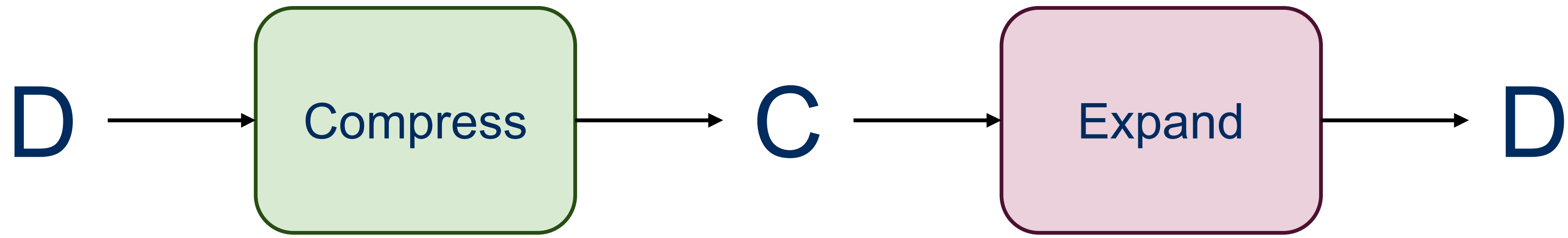
Problem of the Day: Compression

- Input: A sequence of characters
  - *n* characters
  - each encoded as an 8-bit Extended ASCII
- Output: A bit string
  - of length less than 8*n
  - the original sequence can be fully restored from the bitstring

ASCII Encoding

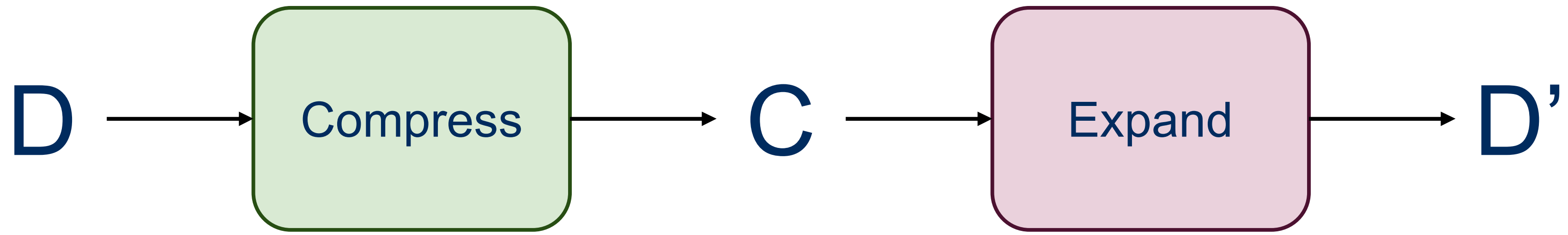


Lossless Compression



- Input can be recovered from compressed data exactly
- Examples:
  - zip files, FLAC

Lossy Compression



- Information is permanently lost in the compression process
- Examples:
  - MP3, H264, JPEG
- With audio/video files this typically isn’t a huge problem as human users might not be able to perceive the difference

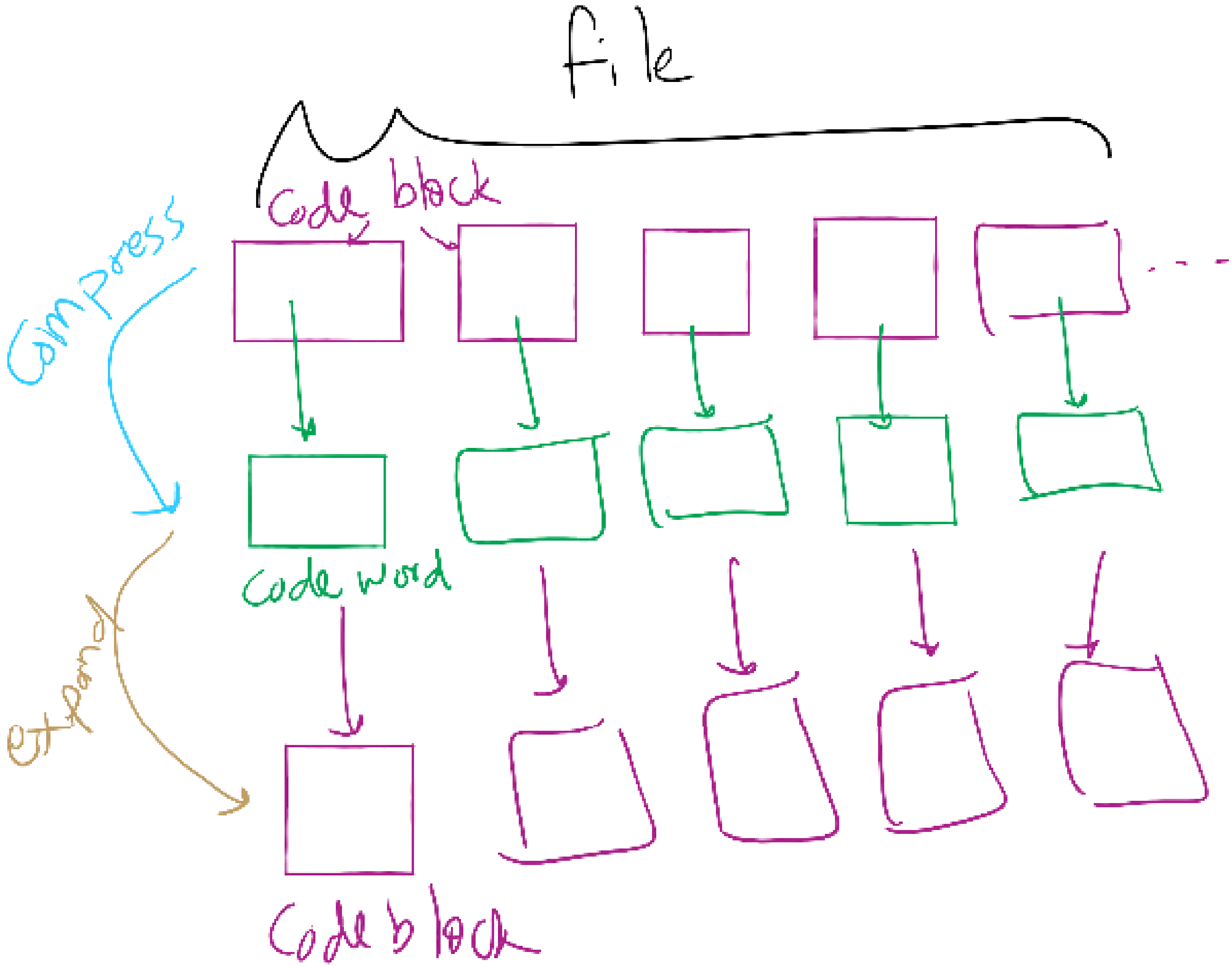
Lossy examples

- MP3
  - "Cuts out" portions of audio that are considered beyond what most people are capable of hearing
- JPEG

40K

28K

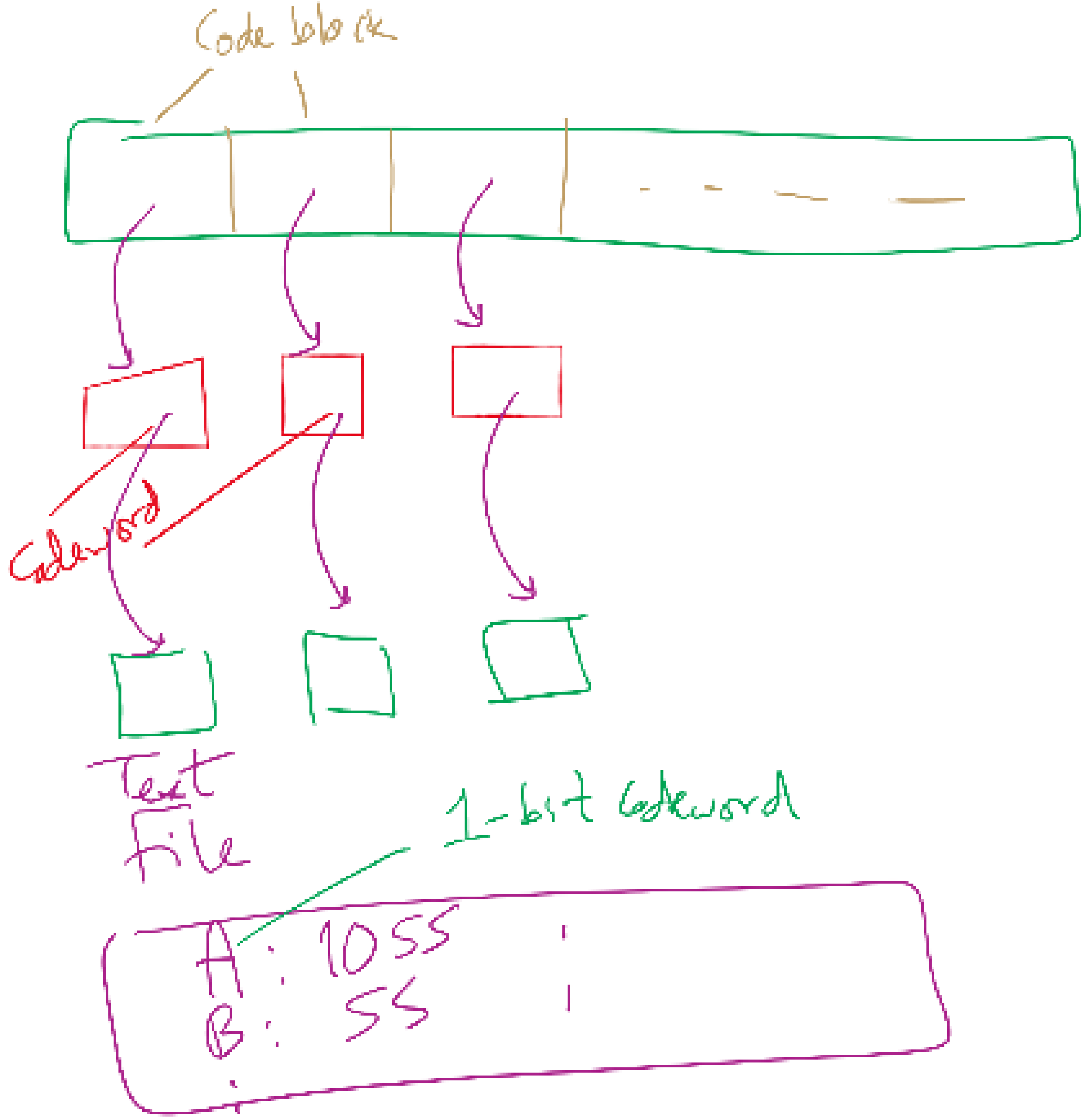
Lossless Compression Framework



Solution 1: Huffman Compression

- What if we used *variable length* codewords instead of the constant 8? Could we store the same info in less space?
  - Different characters are represented using codes of different bit lengths
  - If all characters in the alphabet have the same usage frequency, we can't beat block storage
    - On a character by character basis…
  - What about different usage frequencies between characters?
    - In English, R, S, T, L, N, E are used much more than Q or X

Variable size codewords



But we have to worry about restoring the data!

- Decoding was easy for block codes
  - Grab the next 8 bits in the bitstring
  - How can we decode a bitstring that is made of variable length code words?
  - BAD example of variable length encoding:

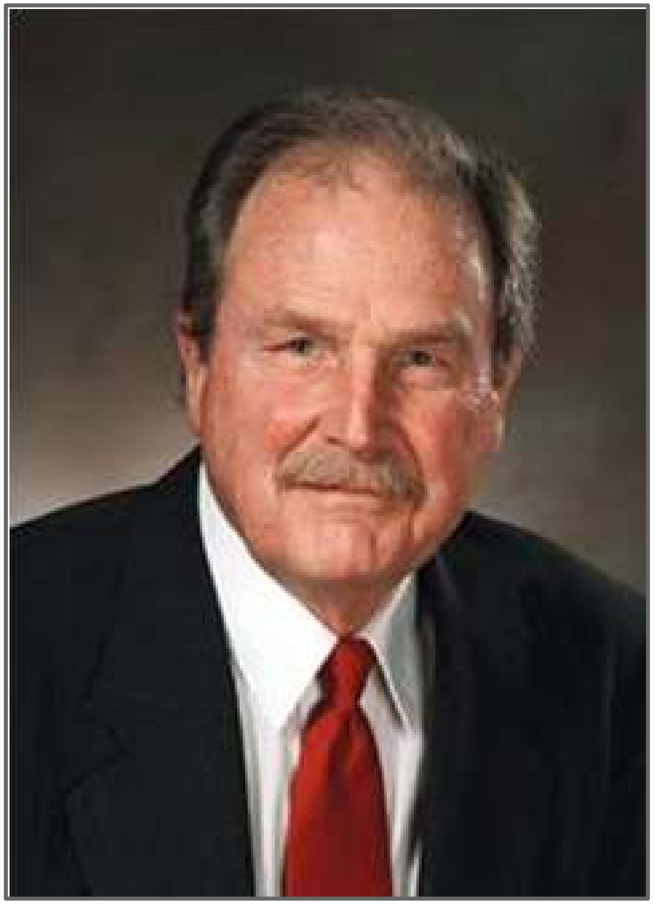
1	A
00	T
01	K
001	U
100	R
101	C
10101	N

Variable length encoding for lossless compression

- Codes must be *prefix free*
  - No code can be a prefix of any other in the scheme
  - Using this, we can achieve compression by:
    - Using fewer bits to represent more common characters
    - Using longer codes to represent less common characters

How can we create these prefix-free codes?

Huffman encoding!



Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com